NASA Technical Memorandum 85784

AN OPERATING SYSTEM FOR FUTURE AEROSPACE VEHICLE COMPUTER SYSTEMS

Edwin C. Foudriat, W. J. Berman,
Ralph W. Will and W. L. Bynum

April 1984

NASA
National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

SUMMARY

The requirements for future aerospace vehicle computer operating systems are examined in this paper. The computer architecture is assumed to be distributed with a local area network connecting the nodes.

Each node is assumed to provide a specific functionality for the space vehicle and as such it can maintain a degree of independence. The network provides for communication so that the overall tasks of the vehicle are accomplished. Hence, the computer architecture and operating system structure implement a "cooperative" autonomous concept which is both flexible and extensible as required for future space vehicles like the space station.

A distributed operating system structure is developed to support the system autonomy based upon the concept of objects. Common service objects that exist at all or most nodes are identified as resource managers (memory, secondary storage, communication, and user interface). The concept for integrating node unique service objects with the common service objects in order to implement both the autonomy and the cooperation between nodes is developed.

The requirements for time-critical performance and reliability and recovery are discussed. Time-critical performance impacts all parts of the distributed operating system from its structure, to the functional design of its service objects, to the language structure used for its implementation. Throughout the paper the tradeoffs -- concurrency, language structure, object recovery, binding, file structure, communication protocol, programmer freedom, etc., -- are considered to arrive at a feasible, maximum performance design.

Reliability of the network system is considered. A parallel multipath bus structure is proposed which enables the control of delivery time for time-critical messages. The architecture supports time controlled message delivery over a wide range of message loading. The same architecture supports almost immediate recovery for the time-critical message system after a communication failure with considerable redundancy to handle multiple failures. Techniques to enable programmer control of recovery as a result of system or service object malfunction are incorporated into the system design.

Because of the freedom provided to the programmer, the user interface structure at program development must contain debug capability. With the understanding that the finding and fixing of errors in concurrent, distributed systems is extremely difficult, debug features are provided at the operating system, user interface level to enable the programmer to monitor and control tasks both within and between nodes.

I. INTRODUCTION

The change from central to distributed computing (networks) taking place in ground systems can be extended to future aerospace vehicles. Networking may be especially applicable for evolutionary systems like those proposed for the future space station. [1, 2] Many of the requirements, and hence, the features, of distributed ground-based systems will be adaptable with modification to their aerospace counterparts. However, since many of the space-borne utilizations and applications differ significantly, the a posteriori modification of primarily ground based systems is not usually feasible.

The task of developing software for a space system is enormous. [2] In this article we will attempt to examine only one small but very critical aspect of that software, the operating system (O/S) for the distributed network. It is small because the operating system uses only a small portion of the computer resources onboard; critical, because the basic assumptions about the network system structure and operating system characteristics have a significant effect on the remainder of the software system.

The objective of this paper is to examine future aerospace computer operating systems software, how this software will be influenced by networking and distribution of tasks, and how it can be made flexible to the evolution (explosion) of future computer capability. In this task, it is necessary to make some initial assumptions bounding the character of future aerospace computer systems in order to clarify the structure of their O/S features.

A. Future Aerospace Vehicle Computer Systems

A NASA critical need for future aerospace vehicles is for highly reliable, time-critical, semi-autonomous computer systems. Autonomous subsystems aboard a future space station will provide attitude control, environmental control, navigation and guidance, imagery, etc. The basic structure and many of the hardware devices for a particular subsystem will be germane to that subsystem. However, to function viably in the completion of specific system tasks, these subsystems must be suitably linked to cooperate with each other. The fundamental question is the implementation of this cooperation, that is, how the general purpose components should connect the special purpose components together to achieve "cooperative" autonomy with maximum flexibility in development, operation, and extension of the capability of the spacecraft.

Figure 1 illustrates the concept of "cooperative" autonomy for some typical spacecraft subsystems. It can be seen that the subsystems themselves may be local networks, e.g., the stability and control (S&C) system with distributed components such as actuators and sensors. The subsystems cooperate by sharing information and resources via data busses which make up the network. For example the S&C subsystem, even though it performs its own function in an autonomous manner, also provides spacecraft attitude and orientation information to an onboard experiment. With proper authority the experiment can command a specific attitude profile. This paper is concerned with the time-critical

network components and software which implement this information sharing and command capability.

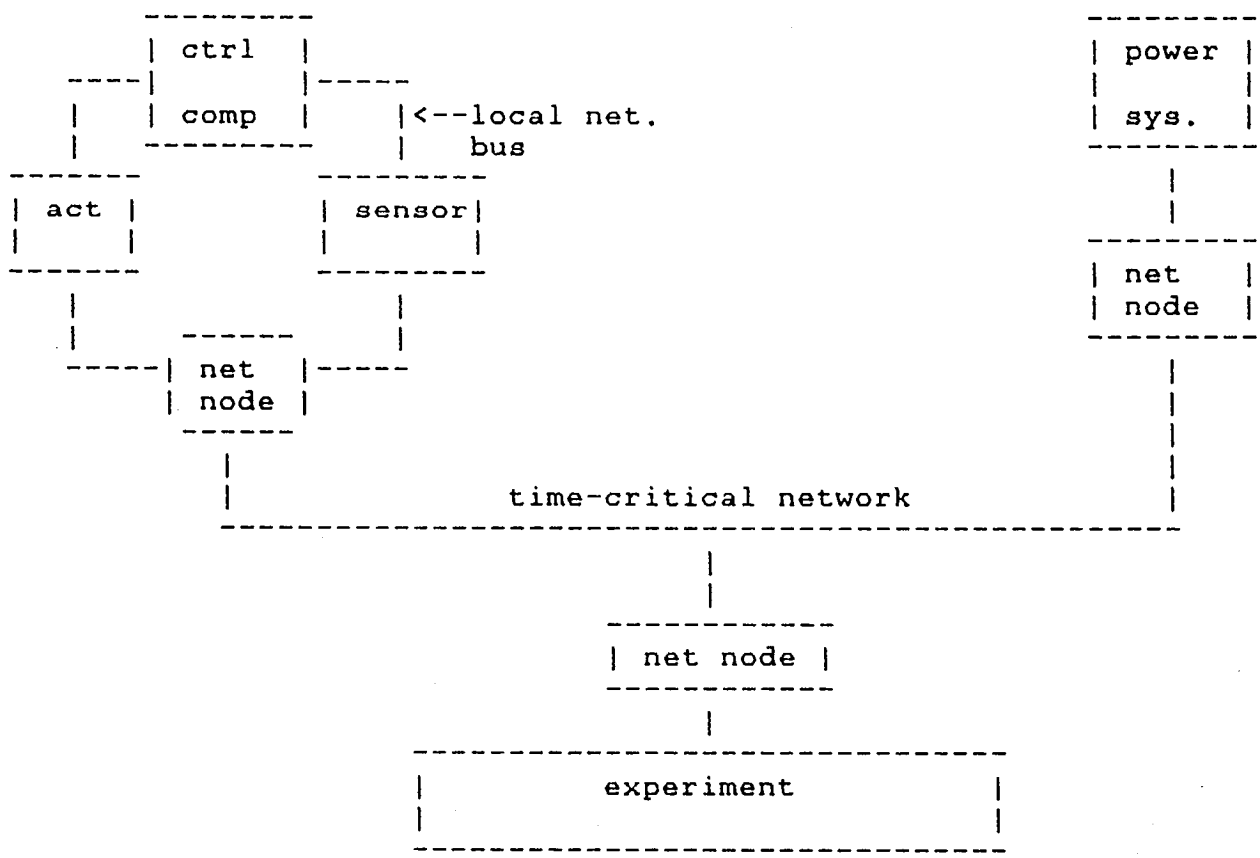

Figure 1.- Cooperative Autonomous Network Structure

There is considerable freedom in choosing this interface. For example, the interface could allow the complete autonomy of each subsystem. Here, each subsystem defines its information content and structure at the network interface. The subsystem side of the interface then is strictly controlled by the designers of that subsystem. If this freedom were permitted, it is possible that additional or modified information at the interface could be extremely difficult and costly to obtain once the subsystem were completed. Alternatively, each subsystem could be required to use common computer hardware and software, making modification to system information easy at the cost of performance (present and future). For a space station, the cost of supporting 15-20 year old hardware architecture needed for the sake of standardization or commonality is clearly unacceptable. Hence, the interface is a compromise which permits specific component functional integration into the system while identifying the common elements between subsystems.

For our design, the space vehicle system is assumed to be implemented with each autonomous subsystem occupying a computer node. All nodes have general purpose computing capability. Autonomy is

obtained as a result of special purpose peripheral and/or computational devices (e.g., inertial platform) integrated at a node. The nodes are linked by a parallel multipath network. There are two levels of parallelism for redundant operation. First, there are parallel busses so if one link should fail another will quickly replace it, at least for the critical communications. Second, there are multiple sets of parallel busses, so that communication between nodes is available along different routes. [3] Nodes can be grouped by association; e.g., those most frequently communicating are placed on the same bus.

There are general features of the system which need to be considered in the design. They are:

a. Time-critical performance - during the actual execution of a task, speed of computation and utilization of concurrency to enhance that speed are extremely important. Most design tradeoffs will weigh speed most heavily. Also, in a time-critical system some mechanism must be available to attain message delivery within a specified time, at least for the class of messages that are critical.

b. Transparency - transparency implies that lower level implementation features are hidden from the user. At the network level this means that a distributed system would appear as though it were a single computer to the user. This is an important feature in many distributed systems [34] where the user wishes his job completed but does not care how it is done. It is given a lesser weight in our system when tradeoffs with performance are concerned. Since many nodes provide unique services, transfer of computation to another node is generally not feasible. Also transfer of computation may result in subtle but important changes, so transparency in this system may be limited to situations that are not time-critical.

c. Target design - The system is intended for use in a space vehicle like a space station. In this environment, it is assumed that very little program development will take place in space because of dollar cost of operation. Hence, the design investigated here is not targeted to the final operational system environment but rather to the program development environment, which contains both the final system and the capability to develop and debug code to be run in the final environment.

The most critical design goals are to define:

1. An interface which will support the operation of semi-autonomous systems (special computations and devices) within a common network system structure. This interface must be both extensible and flexible.

2. A structure that will support reliable, time-critical performance within and between nodes which make up the system.

B. Review of Related Work

During the last few years, the volume of work related to distributed and networked systems has been extensive. [4 - 18]

Features relating to distributed systems include languages [6, 7, 8, 16, 18, 26, 27, 29], operating systems [9, 10, 17], data bases [11, 12], as well as networking, protocols [13, 14], and information systems. This document is not meant to be a survey of the literature in distributed computing. Instead, we intend to discuss the salient features of our design in comparison with current related research.

The language feature most prevalent in the literature for distributed systems is the remote procedure call (RPC). [6, 14, 16] This, coupled with a message system to support RPC bidirectional communication, is the basic structure enabling distribution. A widely used alternative is the send/reply (signal/wait) protocol for communicating sequential processes. [16, 35] While one of these two features appears in virtually all distibuted system implementations, the implied structure behind them and the resultant efficiency differ drastically.

One structure evolving from distributed database research uses objects and actions. [11, 12, 15] Objects are encapsulations of information, i.e., storage containers, with well defined mechanisms for manipulation of the data within. The precise features of an object (location, replication, entry, defined activity, locking, etc.) differs among users. Similar encapsulations [17], e.g., Guardians, [6, 18] have been employed by other reseachers. Actions (transactions) represent tasks or units of work which via their agents (processes) manipulate objects. Generally, actions are atomic when viewed by their user in that they are indivisible; i.e., either the total action takes place and a new system state results, or the action aborts and the initial state remains (or is restored, depending on viewpoint). Splitting the atom has become common practice among Computer Science professionals in that nested actions, subatomic actions, actions within atomic actions, etc., are readily considered. [12, 15, 19]

Actions naturally lead to the question of abort (the inability to successfully complete the top level action) and recovery. "Recoverability" as stated by Allchin [12] "is sufficient to guarantee that the system can maintain totality of an action at all times." As Allchin notes, nesting of atomic actions makes recovery eminently more difficult. In database research, recovery after abort is viewed as replacement or rollback to the state existing prior to the beginning of the top level action. Other researchers dealing with recovery and reliability of computer processes have looked at recovery in a somewhat different manner [19, 20] and have evolved the concept of conversation blocks. [21] Conversations restrict the transfer of information and hence the domino effect of rollback, to those processes explicitly involved. If an abort takes place and rollback is required, the data needed to restore the initial state is defined by scope and the rollback point is strictly specified. Hence conversations have the effect of reintroducing atomicity into the fractured atomic action. [20]

The concepts of recovery and synchronization interact in subtle ways. To quote Allchin [12] : "The amount of concurrency may be limited by the choice of recovery." Thus, one needs to deal effectively with both process and atomic synchronization. Most systems assume, because

of the user's inability to deal with concurrent communicating processes or his difficulty in locating concurrency errors, that actions need synchronization built in for protection. For example, Guardians are basically for the naive user in that no user programming for synchronization and recovery of objects is possible.

The idea of built-in protection, whether due to user ability, difficulty in error detection, desire for transparency, or whatever the reason advanced, prevails in the distributed systems field to the point where very little research has been devoted to distributed debugging. References [22, 36 - 39] are the only activities known to the authors. Only the CHILL program development environment [39] has implemented debugging tools into a real system.

Debugging concurrent tasks is a frustrating, manpower-intensive job. Even in the simple case where a single CPU is interfacing its device drivers in an interrupt driven mode, debugging is difficult at best. Error conditions which depend upon sequencing can occur in a transient fashion and can be extremely hard to capture. The imposition of trace or dump information may change timing sufficiently to mask the error completely. In a system where performance is desired so that programmers have the capability to trade off concurrency for transparency, distributed debugging in the programming environment is necessary.

This survey, while extremely limited, does show the central theme of the basic design philosophy for our time-critical operating system and illuminates where this system differs from the systems investigated by others. Here objects are used to encapsulate and share information. Since performance is foremost, minimal synchronization and recovery restrictions are imposed on objects by the language or operating system. Alternately, the ability for the programmer to provide the most effective synchronization and recovery mechanism for his particular situation is emphasized. To enable him to use this freedom, thread of control tracing of object use and extensive distributed debugging capacity are incorporated into the functional design of the network system.

C. Format of the Paper

In the following section of the paper, the structure of the operating system is discussed. The level structure of the O/S is considered along with the general system components that must exist at every node in order for the network to provide the systems interface. The technique used to integrate the unique features in the form of objects into the general O/S structure is considered so that nodes can provide unique services and hence cooperate autonomously.

The need for language structure and the selection of appropriate language features needed to implement the operating system is then considered. The importance of specific features on performance is stressed. In the last section of the paper, the four general O/S components, storage management, memory management, communications system, and user interface are discussed. A functional design for each is presented and the interfaces between components is considered. The

mechanization of each to fulfill the basic requirements for time-critical, reliable performance is stressed.

II. STRUCTURE OF THE OPERATING SYSTEM

The basic structure of the operating system is shown in Figure 2. The first portion of the operating system is very conventional. The kernel provides both the basic language support used to construct the remainder of the O/S and the run-time support for standard intra-node features. The device drivers map the peripheral device hardware to the system and handle hardware interrupts. Device drivers have been separated from the O/S support layer in the Figure 2 to illustrate that they are hardware configuration dependent. Most hardware drivers at this level are closely allied to the specific O/S resources to which they relate, e.g., device drivers will exist for the console connected with the user interface, for the mass storage devices under control of the storage management system, etc. Structurally they are integrated into the O/S support layer subsystem to which they are related.

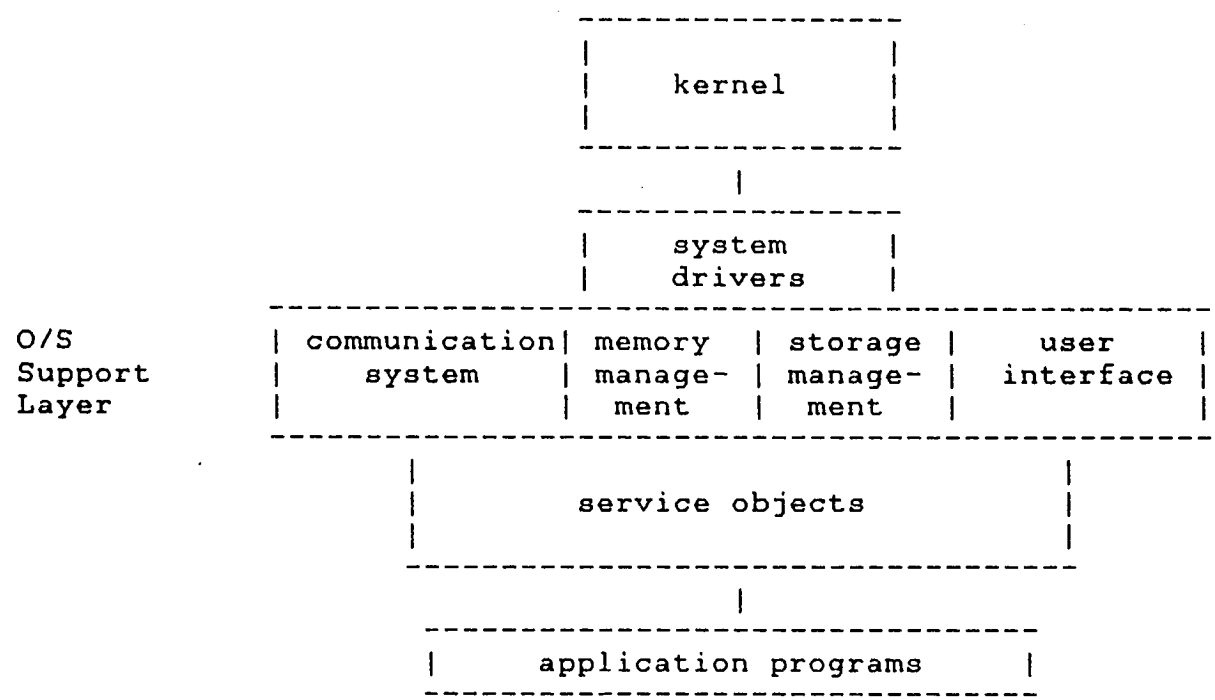

Figure 2.- Operating System Structure

The heart of the operating system is provided by the O/S support layer, which for performance is integrated as a single layer. This differs drastically from vertically layered operating systems [10, 25] and the standard ISO reference model which has seven layers of abstraction to support the total network protocol. [46] The O/S support layer is designed to handle all resources which are common across the nodes, i.e., communication, primary memory, secondary memory and user interface. A major portion of the paper will describe the requirements

and functional design of these four resource managers and the decisions made in order to support time-critical performance and interface to the node specific subsystems.

A few observations are in order concerning the choice of an integrated structure over a layered abstraction structure as noted above. Abstract layers require an interface between each level and each level hides the details of how services are actually implemented. For each level to perform independently and at its own pace buffering is required between layers. Buffering implies copying and the cost for copying is degraded performance. In order to minimize the need for copying, a minimal layered structure is selected and services are shared in an integrated fashion. The price for this structure is that changes in functionality or format of an O/S layer service impact other service elements. Spector [47] reaches a similar conclusion as justification for integrating his communication primitives.

The next layer supports service objects. Service objects incorporate the node-specific services which exist for a particular node because specific hardware devices are located there or computations can be efficiently implemented at that node. Hence, nodes will derive their autonomy based upon the nature of their service objects. Service object features will be detailed in section IV, but in general they are objects in that they encapsulate information and have specific mechanisms to manipulate their internal data. In essence, one can envision service objects as part of the O/S support layer since the support layer subsystems are constructed in a manner identical to the node-unique service objects. Thus, Figure 2 indicates that the autonomous system-network interface exists at the O/S support-service object layer of the node. In the final layer, the application layer, programs use common and unique service objects to complete specific tasks or activities implemented by the spacecraft distributed computer system.

Unique features of this design are flexibility and extensibility. Common hardware is needed only for the communication system protocol and its interface to the CPU. Identical CPU hardware is not needed as long as the structure is capable of handling the software kernel, device drivers, and common O/S support service objects. Common O/S structure allows the nodes to function in a distributed manner and allows new nodes to integrate easily into the system. A unique feature which enables good performance is that special node service objects are readily incorporated directly at the O/S resource support layer. Hence, in the development mode a new node (service oriented or general purpose) can be brought on line easily once the kernel, device drivers, and common resource service objects are targeted. Once a node is on-line, its special device features and hence, its "autonomy" can be implemented.

III. LANGUAGE FOR THE TIME-CRITICAL OPERATING SYSTEM

Most recent operating systems are developed within the context of a programming language. [29, 40, 41] Features of that language play an important part in the features of the operating system. While it is not impossible to implement features diverse from those supported by the

language [24], sometimes these features are cumbersome or the utility of a particular feature is not apparent if not language-related.

Since performance as implied by user-implemented concurrency is the foremost objective, the language should support concurrent tasking and complex synchronization under programmer control. Since information encapsulation is critical for implementing resource sharing and services, objects are necessary. However, objects should not cause undue restrictions on their user. For example, objects in Clouds [43] have action-related items specifically for recovery built within their structure. While this concept may be suitable for many objects in our operating system, at this point it is best that recovery control be left to the programmer. Since the system is distributed, some form of remote procedure call with the inherent restriction of data scoping and implied message generation is necessary. Since the O/S structure is large, the language should support separate compilation. Since the operating system structure is not completely decided, the language should be modifiable so that new or altered syntax and semantics can be implemented when common linguistic properties are identified. Finally, the language should be transportable.

Modula 2 [28, 29] comes close to meeting these requirements. Its major weaknesses are its lack of complex synchronization structures and of syntax that distinguishes between remote and local procedure calls. Ada would meet the requirements except that it lacks an explicit distributed structure. This causes all process calls (local or remote) to be messages, which in turn causes message handling to be implemented at the kernel level. [32] This coupled with Ada's complex thread of control [7] makes tasking expensive to implement, reduces performance, and makes it difficult to transport. Rigid control (no subsets or supersets) makes Ada difficult and costly to employ in an experimental mode.

The language chosen for the implementation is Path Pascal. The major justifications include the unique features discussed below and its flexibility in an experimental environment. Most of the features needed have been investigated for incorporation into the language. [8, 25, 30, 31] Source for the system is easily available, the compilation stages well known, and because Pascal is widely used, competent programmers are available.

Path Pascal provides unique, advantageous features for constructing a distributed operating system. [8] First, the language is object oriented. Particular care has been taken to construct distinct features for the two major language uses for objects, encapsulation of information at compilation and runtime. MODULES provide objectivity for separate compilation. Complete Path Pascal structure, i.e., constants, data types, runtime objects, variables, and subroutines, are available externally simply by naming within an interface section or can be hidden for local use only by declaration outside the interface section (package concept in Ada [29]). Sharing of variables globally and subroutine calls at any level are implemented totally at link time using fairly standard naming techniques so modules create no additional runtime expense.

The runtime encapsulation is OBJECT. Runtime features provide access to the data in objects through controlled entry only. Objects, as opposed to processes [29] or modules [27], contain synchronization using path expressions. Since objects are a Pascal data type, all the Pascal data capability, e.g., nesting, static or dynamic instantiation, explicit or implicit naming, etc., are available. Because of nesting, the problem for path expressions, noted by Andrews [16] "(they) provide elegant notation for expressing synchronization constraints operationally, they are poorly suited for specifying condition synchronization," is not applicable to Path Pascal. Using nested objects and scope rules, condition constraints are easily implemented as variable test conditions and lower level operational constraints. With explicit naming, extremely complex synchronization is readily constructed.

The service object, suggested in this paper as the basic O/S mechanism for distributed autonomy, is a direct extension of the Path Pascal OBJECT with different scoping rules and access related to remote calls. This feature is provided by adding REMOTE syntax to an object. [8, 26] Compile and/or runtime binding are readily incorporated. Hence, the construction of common and unique objects at a node for use by it or other nodes in order to create a distributed "cooperative" autonomous computer system is a direct and logical extension of the Path Pascal programming language.

Research into language structure has provided additional features in Path Pascal that are useful in time-critical systems. DEADLINEs [31] are used very successfully in an environment where time critical activities are important. Also, the storage of initial states (caching concept) in DEADLINEs provides a convenient mechanism to implement rollback for recovery. CAPABILITY [25] is easily implemented in a Path Pascal environment if security is needed.

The ability for objects to be remote or local and hence their entry point calls to be distinct provides an additional feature which may significantly aid O/S structure and performance. In Path Pascal processes are the mechanism for implementing concurrency. Local calls are implemented without implied message traffic. Therefore, the language structure is concurrent at the kernel level so that a local process can be used in a driver, for example. The concurrency is implemented without a complex pseudo message system necessary at the kernel level (as in Ada), so processes can be recursive, appear at any level, are effectively synchronized by sharing objects, and are implemented and context switched rapidly through simple kernel calls (30-50 assembly language statements). Conversely, if the object is truly remote, then remote calls (procedure or process) use the communication system, Figure 2, at the O/S support layer. The implementation at this level will be discussed later.

IV. REQUIREMENTS AND FUNCTIONAL DESIGN OF THE O/S SUPPORT LAYER

Figure 2 shows that the four node subsystems that make up the common O/S deal with the four resources that are common to every (or most) nodes. The following sections will discuss the requirements and functional design features of each subsystem.

A. Storage Management Subsystem

The storage management subsystem, or file system, manages the permanent storage and use of information created by the distributed system. This includes the services provided by the file system as well as the demands made by the file system on other system components. The following definitions are necessary:

file	"virtual memory segment", i.e., completely unstructured or flat. Addressable units (bytes) labelled 0,1,...,n-1.
disk	Generic direct access storage device. Conceptually flat with each block addressable by number: 0,1,...,n-1.
device	Physical peripheral unit.
volume	Fixed or removable medium.
on-line	File is located on a mounted volume, i.e., volume is on some device.

1. Requirements

The file system will provide for storage and retrieval of information created by experiments, of status logging, and of other operational data involved with operating the space station and will provide for program development, execution, testing, etc., in the ground environment. Separate file structures, designated as RT (real time) and TS (time share) respectively, are provided.

2. Architecture

Several approaches are appropriate for satisfying both the RT and TS aspects of the file system. The two most viable approaches are shown in figure 3. The first approach assumes two distinct physical filing systems each with a separate server. The second approach employs a layered structure with one physical filing system and one or more virtual filing systems. These two architectures are functionally equivalent and offer similar advantages.

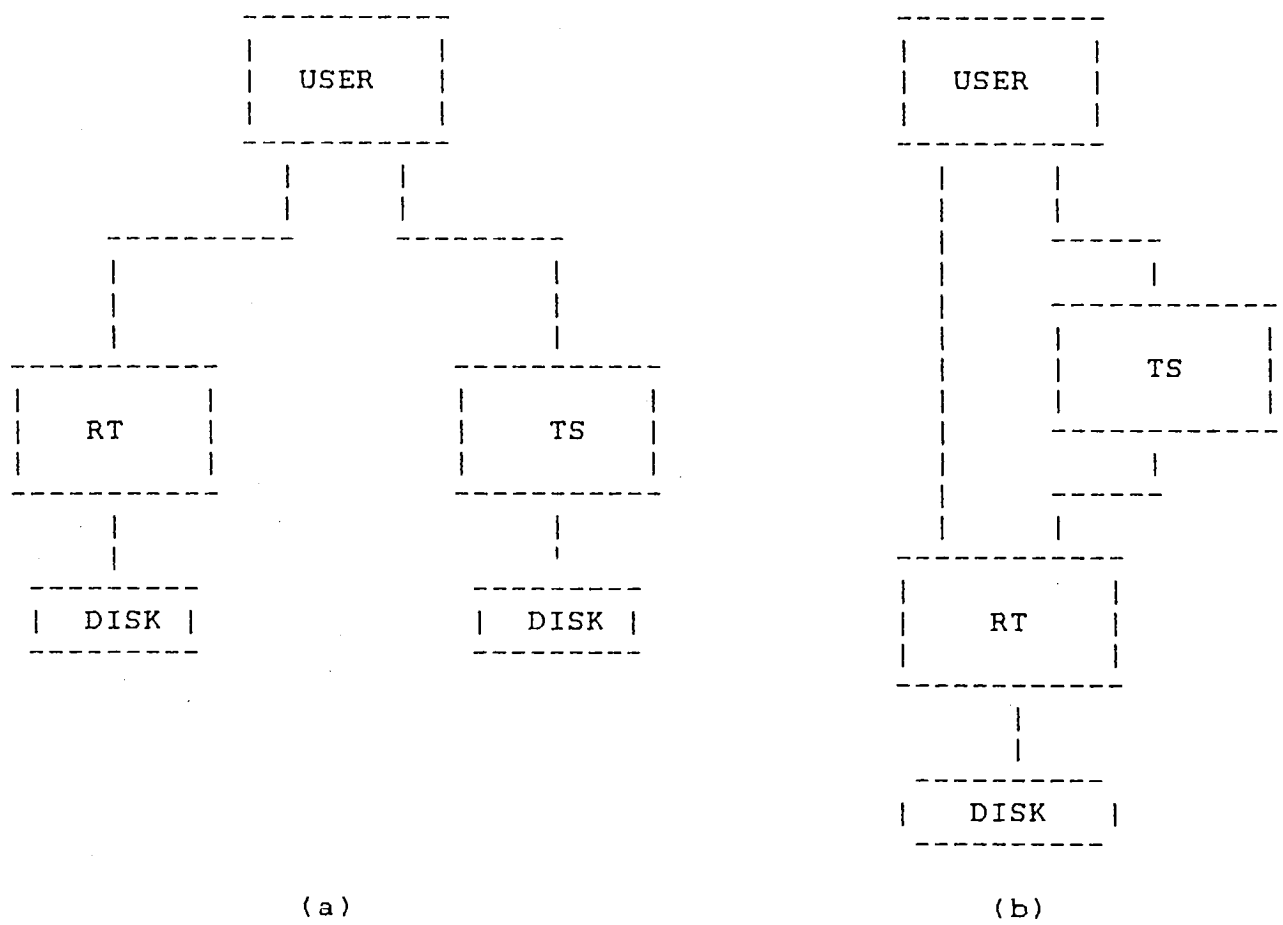

Figure 3.- File System Approaches

Specifically, the advantages of either file system approach are the following:

a. The RT server is brought up early and used for all applications including system development.

b. The TS server is developed on the early prototype system. It is implemented and tested on a noninterfering basis.

c. When the TS server is available, all appropriate files are moved from RT volumes to TS volumes.

d. The experience gained from implementing the TS server is used to implement other filing systems or filing system emulators.

The second approach, figure 3b, offers the following additional advantages:

a. The TS server is smaller, and because it interfaces with the virtual disk defined by the RT server, it need not contain device drivers. The TS server is also more device independent.

b. The disk format of all volumes is the same at the lowest conceptual level. This offers the opportunity to write one set of common utilities (e.g., file dump and garbage collection).

These advantages which accrue while retaining the performance desired for RT applications and sufficient performance for TS applications, are reasons to favor the latter approach.

3. Name Server

The basic function of the name server is to provide information about the services and data available across the network. [23, 33] The reason that this information is provided by a name server rather than a sophisticated information search at run time is to speed up the execution of running programs and services. Hence, the name server is a sophisticated global system directory.

The name server provides information for a number of program activities. For the compiler, it provides the location and access information for the module interface. Information concerning module(s) accessed across the network will be retained by the compiler to be used for consistency checking at load time.

The name server responsibility at load time is to check the consistency of the information requested, i.e., has the module used been modified since it was used by the compiler. Since modules can use other modules, the name server must provide the location and access information for these secondary modules. The information is provided so that the loader can formulate load request messages.

The name server requirement for the edit function is to provide information concerning location and access rights of the file requested by the editor. The editor can then formulate file transfer request messages to access the file for its intended purpose.

Finally, the name server provides runtime support for the Path Pascal IMPORT standard procedure. [26] While it is intended that most execution will use compile-load time binding for execution speed, the capability exists for runtime binding using IMPORT. The name server provides the location and access information of the requested service to the IMPORT . The IMPORT routine then takes the actions necessary to access and use the requested service.

The major name server problems are that of; a) defining precisely the information needed by the name server to fulfill its required functions, and b) maintaining consistency across the nodes of the network in a timely and efficient manner.

4. Name Server / File System Interaction

The name server is a critical element in the operation of the system. It provides the location of files by unique name throughout the system. To access an on-line file, F, a request of the form (Filename, byte) must be mapped to (node, device, volume, file, offset). It is

assumed that the network name server maintains information about the current network configuration. The specific information of interest here is:

volume V is mounted on device D at node N.

The name server can be used by the file system to ascertain whether V is mounted and if so, the user can be connected to node N. It can now be determined whether F exists. The precise nature of the file system and name server interaction can best be explained by means of an example. Figure 4 shows a conceptualized sequence of events resulting from the execution of the file system command

open(V, F, ...)

by an application program. The arc labels indicate the order in which communication takes place.

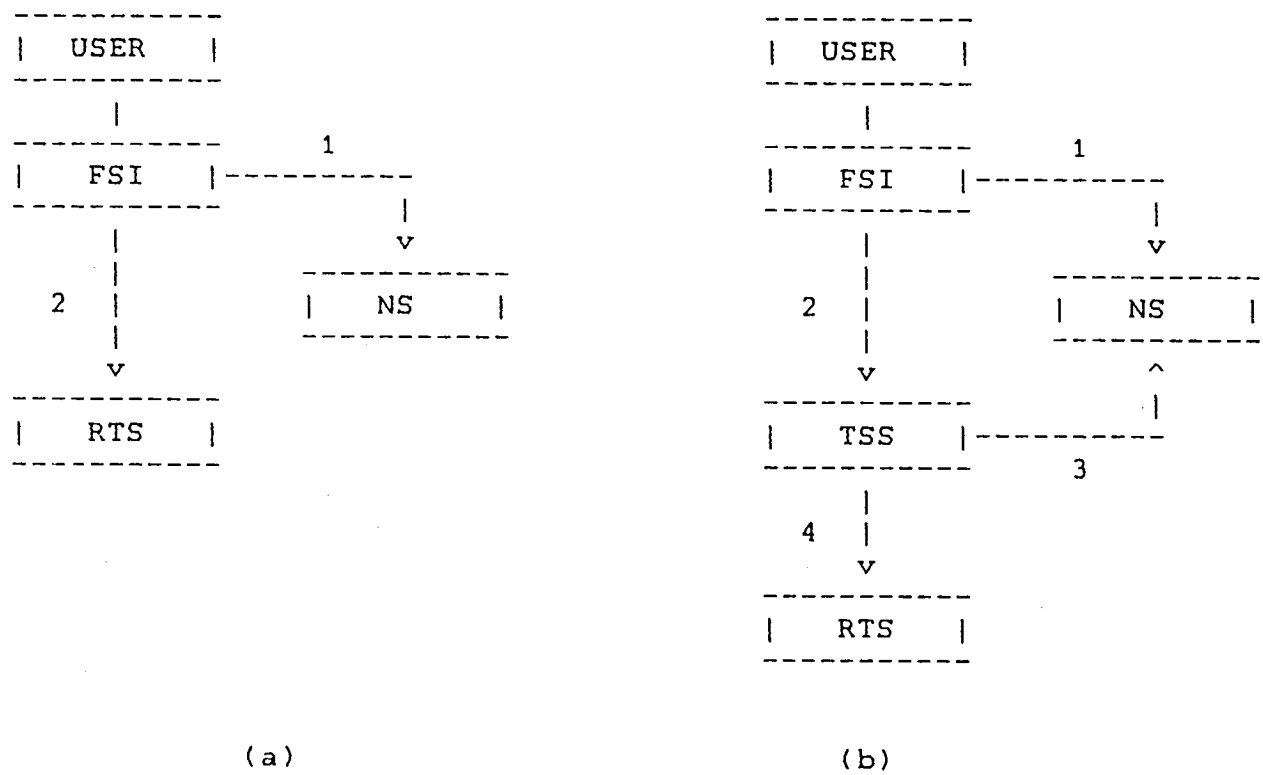

Figure 4.- Sequence of Events Opening File

In figure 4a the "open" command refers to a RT file. This is implicit by the parameter V. The file system interface (FSI) interprets the open command and requests the appropriate RT server (RTS) from the name server (NS). The name server locates (or instantiates) the proper server and establishes the communication link. Next the file system interface asks the RT server to open F.

Figure 4b shows the effect of an "open" on a TS file. (Again, the file type is implicit in the paramater V.) The sequence of events is exactly the same as above up to the request by the file system interface asking the TS server (TSS) to open F. At this point, the TS server is aware of the fact that V is a virtual volume and maps V to V', the appropriate RT volume. A request is then made of the name server to locate the proper RT server. When the RT server is made available, the TS server can open F using the RT server's capabilities.

The communication links established by the name server remain in existence until a "close" is issued by the process holding the link or until some unusual event takes place. In the case of a failed "open" or "create" command, the file system can request that the name server remove the links. In the event of a crash, the name server must assume the responsibility of removing or recovering communication links.

Although not shown in the figure, it is clear that the name server communicates with the loader, which in turn has access to the file system. This implies that at least one file server must be established at cold boot time.

It should be noted that Figure 4 is not intended to imply any specific system topology. In particular, there may be more than one name server and the indicated processes may run on one or more processors.

5. User Interface to File System

In dealing with a TS file, the file system interface is similar to a conventional non-distributed file system (e.g., UNIX [10, 40]). The main addition is the inclusion of an exception processing capability to handle communication or node failures.

The RT file system, on the other hand, will be quite different in that the user must specify all system requirements (e.g., file contiguity, communications bandwidth, etc.) explicitly during file open or create operations. Furthermore, since the file system cannot guarantee a given performance level during file open or create, a real-time program must open/create all files prior to initiating real-time operations. Deadlock situations will be resolved by using a probabilistic time-out in attempting to open/create all necessary files.

6. File System Services

The basic component of the filing system is the RT file system. This component provides contiguous storage allocation for maximum performance under real-time conditions. In addition, this component defines a virtual disk upon which other file systems can be implemented.

The particular services provided by the file system are the following:

a. A real-time file system providing contiguous storage allocation.

b. A time-shared file system (built on the RT system) providing hierarchical directories and dynamic storage allocation (probably similar to the UNIX file system [40]).

c. A uniform interface to all file servers (e.g., RT, TS and possibly others) which is independent of the underlying storage techniques. This interface will provide the usual file manipulation commands (e.g., create, delete, rename, copy, move, etc.) as well as a common set of file access commands (e.g., open, close, read, write, etc.).

d. Files can be opened in read, write, append, read/append or update (read/write) modes. In general, files may have multiple readers and/or multiple appendors, or a single writer.

e. The user may specify the degree of urgency (i.e., access is time-critical) when a file is opened, since continuous I/O in a real time environment demands high performance. In the limiting case, a device may have to be dedicated to a process in order to achieve sufficient performance.

f. The user may specify the degree of desired contiguity when a file is created.

g. A facility for redundant storage will be provided.

B. Memory Management Subsystem

The memory management system consists of (1) the linker, which prepares programs and service objects (subsequently referred to as tasks) by combining the code for local modules and preparing link tables for the remote service objects; (2) the loader, which assigns memory spaces for the code and data for the local task and which establishes that all called remote service objects are installed on their assigned node; and (3) the runtime system, which controls the execution of the tasks by handling, in conjunction with the communication and file systems, the control and transfer of information between the different nodes.

1. Requirements for the Memory Management Subsystem

Linker

The linker creates executable modules from the translator and assembler segments of Path Pascal compiler. These modules contain linkages to all permanent runtime support such as the kernel and the O/S support layer service objects, and to a table containing linkages to all remote service objects, including entry points and the information formats for the input and return parameters. This information is made available at the MODULE level at compile time through the name server and file system.

The linker requests originate from remote or local sources. At present they come only from a programmer's terminal activity, but

consideration is given to linking based directly on program activity.

Loader

The tasks accomplished by the loader are the following:

a. to load code and reserve data space in memory for execution of a task.

b. to release code space from a dormant task in order to load more critical code. Future consideration will be given to reusing, archiving, and reloading of data space, although the present Path Pascal implementation requires that data reoccupy its space in core. Therefore, this feature may be implemented on systems where hardware implemented virtual memory management is available.

c. to overlay code upon suspension and reassignment of code space.

d. to provide for loading all service objects prior to execution.

The loader requests come from a programmer specifying the execution of a program or from another program's load request to ascertain the existence and availability of service objects at a particular node. The former comes from programmer terminal activity; the latter is a result of programmer activity through the communication network (local or remote) and is formulated by the loader at the programmer's node.

Runtime System

The runtime system supervises the execution of tasks. It handles call/return activity to service objects on remote nodes by constructing the necessary information for the transfer. In addition, the runtime system maintains a thread of control between programs and service object calls. This information consists of caller/called links and the status of the call and is sufficient to trace the system state and to provide for recovery of the system to an acceptable state.

The runtime system requests come from the loader, which begins the execution of a particular program or initializes a service object, or from the communication system, which handles message information to and from other nodes on the system.

2. Functional Design of the Memory Management System

Linker

The basic objectives of the linker are the following:

a. To provide an executable code file and the necessary information for the loader-memory management unit to subsequently handle the object code. The linker provides the following:

1) The total object code length for this task.

2) The total data length required for this task.

3) All entry point information for service objects. This includes information to locate the entry point code source and to format the input and return information for the runtime manager when a call/return occurs. The linker provides all links to the runtime kernel and standard runtime objects.

4) The ability to relocate code.

5) Any additional information like priority, usage, time constraints, etc., that may be critical to the loader during loading or execution and to the program monitor(debugging) system. At present the debugging system requires program variable symbol table information and statement boundaries to enable variable and execution control of the Path Pascal source.

6) The ability to determine and communicate to the program monitor system any deficiencies in the executable code such as unresolved externals.

b. The linker should obtain its information from the translator and assembler system. This information includes a file with the relocatable code, its relocation points, and its starting address. In addition, this file contains symbol table information on modules separately compiled, variables, remote objects, including their complete entry point information, and any subroutine call points in order that the code can be reloaded in a new location and executed properly.

Loader

The basic objectives of the loader are the following:

a. To load new tasks in the following manner:

1) If the request is for a service object, the runtime status information tables are checked to see if the service object is loaded. If the service object is already loaded, the loader binds the new request to the present service object runtime format including the newly established communication links.

2) If the request is not a service object or is a service object which is not available, the size of the code and data space necessary to run the object is ascertained from the load source. In order to make the loader decisions, priority and time-critical information is needed.

3) Ascertain from the loader memory tables whether sufficient space is available. (Depending upon virtual memory facilities)

4) If data and code space are available, the free space is allocated to the task. If data space is available but code space is not, the runtime status information on each process is scanned (see 2. below). If a task(s) exists whose priority is lower and whose

activity is minimal, code space for the task(s) is tested to see if it is sufficient for loading the new code. If so, the task(s) is deactivated and the freed space is allocated as above.

5) If no data space and/or code is available, the requestor is informed that the task cannot be loaded.

6) If space is obtained, as in 4) above, then the loader fetches the source code from the disk, corrects any relocatable code information, creates the necessary entry point information if the code is a service object, and loads the code into core.

7) The loader creates the information necessary to enable the runtime system to execute the task. This includes creating the entry in the runtime status information table. If the request is to load a program, the loader creates a process descriptor at the base of the data space including the code starting point. The pointer to the process descriptor is passed to the Path Pascal kernel to be linked onto the ready queue system. If the request is to load a service object, the loader formats the service object entry table system. The loader then formats a message to the requesting program that the service object is available and binds in the communication port for message traffic. If service object initialization is needed, the appropriate entry into the runtime status table is made.

b. To perform memory management by releasing code space from a dormant and/or low priority task(s). This is done by scanning the runtime status table for the lowest priority - least used task(s). If priority is less than or equal to the priority of the code to be loaded, the task is deactivated as in a4) above.

c. To provide reloading for tasks which have been suspended. A task's code can be deactivated as in b. above. Inactive code results from task calls to another service object (local or remote). A procedure call suspends the calling program until the call is completed or a time-out occurs. Alternately, a low priority program can be deactivated because of a lack of activity. Upon freeing space at the termination of another task or upon receipt of a return to the calling task, the deactivated task is reloaded. Reloading is implemented in a manner similar to the initial loading, although many of the load time bindings are preserved in the suspension. The reloaded program is passed to the runtime manager to continue execution.

d. To provide for the loading and binding of all service object calls before execution begins in order to gain runtime efficiency. To accomplish this the loader, in conjunction with the name server, formulates and sends a load message to another node requesting that the service object be loaded. The loader will then wait for acknowledgement. If the remote node is not able to load the service, duplicate or redundant service through the name server can be checked and an alternate load request issued. When all remote services are in place the program commences execution.

Runtime system

The basic objective of the runtime system is to support the execution of tasks by maintaining the remote linkage and creating information and control for remote calls.

The runtime system maintains the information status table created by the loader for each task existing on the node. This information includes status, e.g., deactivated, code in core, etc., a pointer to each code and data block, a pointer to each code file on local storage, and other information needed to maintain the activity of each operating unit.

The runtime system supervises the execution of all tasks. In order to execute, a task must be in a ready-to-run status as reflected by being on the ready-to-run queue. In addition, each task contains a ready queue linking its processes which are ready to run. Both queues are maintained on a priority basis. In order to obtain a runable process, the runtime system scans the ready-to-run queue checking each task's ready queue until it finds an executable process. Task processes are removed and added to the queues in the standard Path Pascal fashion. [8]

The runtime system maintains linkage and control for calls to remote service objects. Two activities considered by the runtime system are:

a. at the calling site a) set up the remote entry call including the parameters and b) receive the remote entry return for the calling program and,

b. at the called site a) establish the called entry procedure or process execution and b) set up the return information transfer at completion of the execution.

This situation assumes successful completion of the remote call activity. If the remote object cannot be run, the runtime system handles the completion protocol by sending a return message.

The following structures and protocols are used to facilitate the runtime system operation:

a. At the local node, the remote service call causes the standard activation record and a remote call data record to be established. The specific service object call is identified by location in the remote entry table which contains remote objects, remote entry points for that object, and local port pointer established at load time. The message format, which includes identifiers for the remote object, remote entry, local task, local procedure, unique ID (pointer to the call activation record), and data to be passed, is given to the communication system port. The port assigns and retains a unique message ID by which the return message can be identified. The pending message information is stored at the task stack location.

b. At the receiving node, the port provides the message to the runtime system. The runtime system then instantiates the call entry point on the global service object stack. In order to maintain consistency, i.e., to enable a service object to receive and execute multiple entry calls, the entry point procedures and process calls will take the format of Path Pascal processes (slightly modified) so that they can be effectively suspended. This will permit service objects to handle multiple entries based upon their path expression synchronization protocol. The message data is then placed on the called entry point stack and the process is placed on the queue.

The runtime system also stores the calling message information in order to make the return call. Upon return, the remote task designation, remote entry designation, local service object, local entry point, message number, and data are formatted into a communication message and shipped to the orginal site.

c. When the return message is received, the message is passed from the port to the runtime system. It transfers the data to the proper activation record, including the setting of any value-return data. The standard return from a call is executed and the process put back on its ready queue.

The final task of the runtime system is to free data and code space at the termination of the task call. On termination, the code and data space are returned to the deallocated memory table, and the information status table entry is eliminated. Any service objects and communication ports used by the terminating operation are notified of termination.

C. Network Communication Subsystem

1. Requirements

The network communication system requirements can be divided into two portions: those for the network control and those for the message service provided to the other operating system and application programs. The requirements for the network control are the following:

a. The communication system is responsible for maintaining the topology information about the network. [13] This includes all necessary information and routing decisions for the parallel multipath system. The network routing is configured to provide a controlled maximum delivery time in a probabalistic sense, kSIGMA, for a message, based upon the average message traffic per bus. This control should be maintained for a wide range of message loadings.

b. The topology information is capable of dynamic updating, i.e., it incorporates new nodes as they come on line and deletes nodes or links that go inactive or fail. For failed linkages, rerouting of messages is handled by the communication system.

c. The communication system provides an emergency message capability, i.e., the ability to interrupt normal message traffic

including controlled kSIGMA delivery time, to send and receive emergency messages.

The communication system provides a message system capability to the remainder of the operating system and applications tasks. The requirements for this service include the following:

a. To provide message transfer between two nodes in a virtual circuit format. [14,43] The virtual circuit is established at load time between tasks running on different nodes. A method for unique identification and matching of call-return pairs [6] is maintained. To reduce the creation and handling of orphans (unwanted messages which are associated with a crashed task), the message service provides assured single delivery with controlled kSIGMA delivery time when proper priority is assigned. Finally, the message service handles variable length uninterpreted data.

b. To provide a multicast service, the delivery of the same message to more than one destination. [13,37] This service is needed because certain spacecraft nodes provide system status information across the network, hence the need for a one-to-many (or all) service. The message system does not have to maintain information on which nodes are registered for particular multicast information. This information will be kept by the service object source of the status data. Since the service is dynamic, the message system will handle multicast changes. This message service does not provide guaranteed delivery to each node because of the complexity involved in the retransmission logic which is required.

2. Functional Design of the Communication System

To fulfill the requirement that the network system maintain the system topology, each node contains a table which describes the topology of the entire network. This global topology model was selected over a "local knowledge" scheme for reasons of operational efficiency in the environment where message path selection is done at load time. In this case, complete bus selection including gateway nodes can be accomplished within the framework of the local node. The global model requires more storage and a search for the best path at communication request (task load) time, but it does eliminate message traffic between nodes and gateway nodes during path selection.

The following information is maintained by each node:

a. The unique name of the node incorporating its instantiation number to determine when a node fails and is brought up again.

b. A list of busses to which the node is connected and its device number on that bus - each bus designator is 8 bits (256 busses) and the device number is 8 bits (256 devices/bus) making 2 bytes per bus. The list is 10 elements long (each node connects to a maximum of 10 busses) for a total of 20 bytes.

c. Status byte

The system operates in the following manner:

a. A new node simply obtains its global table from a neighboring node, adds an entry for itself, and broadcasts an "I'm here" message containing its own table entry (alternatively, it could broadcast the new tables) so that all other nodes can add it to their tables.

b. A new hookup (new link of an existing node to another bus) broadcasts an "I'm here" message which updates all other nodes' tables.

c. For a dead node, the node detecting the failure through a diagnostic routine broadcasts a notification of a dead node (or new tables) which removes that node from all nodes' global tables.

d. When a node discovers a dead link to a particular bus with the aid of diagnostic routines, it will broadcast a new "I'm here" (or new tables) message which reflects the changed linkage.

The network communication requirements to handle and deliver messages is fundamental to the performance of the distributed system. Especially important is the concept of controlled kSIGMA delivery time for critical time (process control type) applications. However, this is an extremely difficult feature to provide in a network environment. Absolute guarantees of response may be impossible because of the random nature of events on a network; therefore, a statistical estimate of expected delivery may be the best that can be provided. Trying to guarantee message delivery via ring network polling (time slice) techniques can impose a large penalty on system performance and efficiency. Even then, if one node has multiple critical messages to send over a short period of time or if a new node is added, the guarantee is lost or altered.

A more reasonable technique, which preserves system efficiency, uses path selection and runtime monitoring and control of message traffic to insure that a message gets the required service. Routing algorithms assign the route based upon the fewest links. For time-critical messages, the topology system maintains the currently established maximum traffic load on a particular bus. As a result, bus access time is controlled. With this information, the expected delivery time of a message between network nodes can be calculated.

The message traffic on the critical busses is monitored to assure that expected time response can be met. Based upon prior statistical evaluations of bus message traffic verses access time, critical bus access time can be changed (within limits) in order to accommodate a particular distribution that requires different delivery times. Hence, by proper system control and scheduling, kSIGMA delivery is maintained and controlled.

By monitoring the message traffic, periods of slack bus activity can be detected. During these periods non-time-critical messages can be placed upon the critical bus without causing delays in excess of the bus delivery schedule. In this way, the routing algorithm avoids creating message bottlenecks by blindly overloading a particular bus,

and the communication monitor can increase system efficiency by taking advantage of unused time-critical bus capacity.

For system reliability, the network bus structure is redundant. This parallelism is used to advantage by segregating the time-critical messages from those which can tolerate some delay. This insures minimum interference with time-critical communications. Busses are designated as either time-critical or normal message paths with each node linked by at least one of each type bus, and with as much parallelism as feasible between the time-critical and normal bus structure. Routing algorithms will assign busses depending on the type of message.

By assigning parallel critical and normal busses, flexible message delivery control is possible. The critical bus is used to relieve some of the normal bus traffic during periods of heavy normal message activity, if the critical bus has slack activity. Furthermore, during peak periods of critical activity, the normal bus can be used to handle critical messages, first, by a priority scheme and second, by actually assigning more than one parallel bus to be critical. With such flexibility, it should be possible to control expected delivery time effectively for a wide range of traffic conditions.

Finally, an emergency message capability is imposed upon the system. When an emergency arises, any node with proper authority can take over the bus without the normal bus arbitration scheme. By providing a silence signal or other message override, that node causes all nodes on that bus to immediately cease transmission of messages and access control. Then after a suitable silence delay, the node can transmit its emergency message which may be propagated throughout the system or be directed to specific destinations.

Using the above parallel bus configuration, a virtual circuit message system is accomplished by the following communications concept. At load time, a virtual circuit is established between tasks. Each service object referenced by the task being loaded has a separate port. These ports remain for the lifetime of the task and can be used to service all entry points of the remote object. Figure 5 shows a conceptual diagram of the message transmission path.

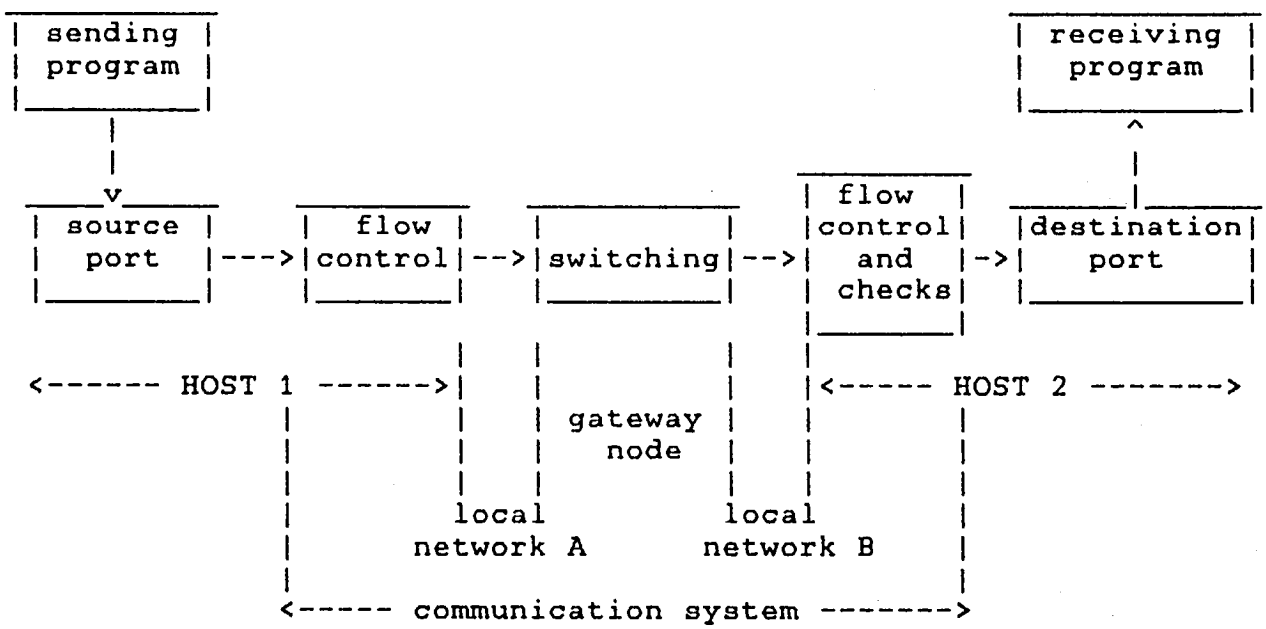

Figure 5.- Message transmission path

The ports are established by a dialogue between the loader and the communication system. The request from the loader is a combined request to load the service object on the remote node and to establish the virtual circuit. Hence, a unique ID for the service object, including its node, is passed to the communication system. Additional information concerning priority, message timing, and message length will be provided. The communications system provides the routing, prepares a message that includes the request for a remote port to be established, and sends the message. Time stamping can be used to check that timing constraints are fulfilled.

Upon receipt of the message, the remote node passes the request to its loader. An affirmative load request return causes the remote node port to be confirmed and the virtual circuit complete. When the initiating node receives the affirmative load message, the communication system retains knowledge of its remote port pair and passes the message to the loader along with a pointer to its local port address. If all load requests are established successfully, the task is transferred to the runtime manager with all its ports ready.

A negative load request return causes the remote node port to be cancelled and a message posted to the initiating node. The initiating node then cancels its port and passes the message to the local loader which must handle the rejection. If alternative task capability does not exist, the loader cancels all its local ports as if termination had occurred. The ports then cancel their remote pair.

During task runtime, the system handles remote procedure or process calls directly with the port. Since all virtual circuit connectivity is established, the port only needs to attach its

predefined header information to the message, packetize it, and place it in the communications channel. The ports provide the necessary sequenced packet control structure to assure single delivery of each message. It should be noted that the remote procedure call is not implemented identically to single machine procedure calls as in reference [48] in that the RPC provides a time-out mechanism. It is felt that the distributed environment differs from that of a single machine since the calling process is independent of the callee and can proceed even though the callee has problems. In the single machine environment, caller and callee are the same machine so this is usually not feasible. At termination the task cancels all of the ports under its control. This in turn cancels the remote port and remote service object.

As a result of the virtual circuit with identifiable ports interconnecting remote services, it is possible to retain a complete thread of control.

In order to control buffer size, a maximum length message is established. For large files, the transfer is done by blocks in which each block is considered a separate message. Blocking will be the responsibility of the task using the message service.

The network is responsible for transfer between busses through gateways with special gateway ports established for this transfer. Since the routing information is available locally, the necessary intermediate gateway ports can be provided in the header constructed at load time. Gateway ports have control over intermediate headers as necessary to route the message.

The requirement for a multicast system is satisfied using concepts which are in many ways simpler than virtual circuits. In the spacecraft system, multicast will be primarily used to provide system status and event notification. It is assumed that each node wishing particular multicast information will make a request to the service object surrounding that information. These may be general node related tasks or specific application programs. The loader checks if that service object information is already at the node, and if not, the loader request is processed similarly to that in the virtual circuit system above, i.e., a receiving port is set up. The port determines the routing based upon the parameters specified. At the remote node, the service request status is checked and if the status information system is active, the new node request is added to the sending port address list.

The multicast address nodes are provided to the service object port at message time. Receiving node addresses are translated to the message header address. Special ports at gateway nodes are provided for the transfer of messages from one bus to another. These ports operate for both virtual circuit and multicast transfers. These ports, which are constructed as the system is created and new busses are added to gateway nodes, accept messages from one bus and output them on (an)other bus(es). For virtual circuit (point-to-point) messages, a table designating the output bus and the eventual receiving node is used. Table entries based upon eventual destination node are

constructed at the time the virtual circuit is created. For multicast, similar table entries are created but are based upon the sending node. The tables indicate to the gateway node which nodes on a particular bus receive the multicast message.

The multicast system, unlike the virtual circuit, does not provide assured delivery of a single copy. Instead it is a one-way system with reasonable best effort to deliver the copy.

D. User Interface Subsystem

1. Requirements of the User Interface Subsystem

Three distinct user interfaces are supported -- experimenter, operator, and debugger. Each successive level of user interface has all of the capabilities of the preceding levels plus new capabilities necessary to accomplish the tasks of that level.

Experimenter Interface

The experimenter interface is the least defined since the requirements are most variable. Support is provided for multiple windows and graphics on output as well as non-keyboard data entry devices such as a mouse. The exact interface requirements must be specified by the designer of the experiment. Therefore, the implemented experimenter interface is designed to support the variability, simplify the development of particular experimenter interfaces, and insure a degree of commonality among all such interfaces.

Operator Interface

The operator interface allows additional system monitoring and controlling. An interactive command language (based upon our previous experience with the Interactive Software Invocation System) [45] gives the operator extensive capabilities to initiate, monitor, and interact with various programs running in the system. This language also allows the operator to perform such common tasks as file manipulation (creation, deletion, transfer, etc.) and direct transmittal of messages using the communications server.

Debugger Interface

The debugger interface allows a programmer to gain direct control over a program running on the system. Using this facility, the programmer can initiate program execution, monitor program progress, "break" the execution of the program based upon the occurrence of sequences of events occurring within the program, interactively interrogate, and, if necessary, modify various data structures within the program, or terminate program execution. Since debugging is a critical aspect of this research project, it is discussed in more detail below.

2. Functional Design of the Debugging System

The debugging problem is to identify and correct programming mistakes. This is particularly difficult in real-time, multi-node, concurrent systems. First, the number of possible execution sequences is, for all practical purposes, infinite. This means that no amount of testing can guarantee that all error-producing execution sequences have been attempted. Second, even if such an error-producing sequence is detected, it may be very difficult to reconstruct exactly. Real-time systems are notorious for producing irreproducible results. Finally, even after an error has been detected, resolving the exact mistake(s) that cause that error can be quite difficult. Locating the cause of an error is an art that depends upon the skill of the programmer and the ability of the debug system to allow the programmer to specify very precisely the information that he believes will be of assistance.

In order to detect and correct both language errors (violations of the rules of the underlying programming language and operating system) and logical errors (violations of the logic required to solve the problem), a variety of techniques is used. In general, language errors will be detected by the compiler or by special error-checking code generated by the compiler. Logical errors must be detected by testing; however, once such an error has been identified, a sophisticated runtime debugger must be available to locate and resolve the error.

The debugger is designed to assist the programmer at the program, server, and system levels.

Program Debugging

The key concept in program debugging is that the programmer specifies a particular sequence of events, using a debug expression, that is used to trigger the asynchronous calling of a Path Pascal subroutine (the debug action). Since the events of which the debugger is normally aware are the entry and exit of subroutines (Path Pascal PROCEDUREs, FUNCTIONs and PROCESSes), a debug expression is a modified form of path expression [44]. Several different debug expressions may be in place at any time, and each of these will have its own debug action.

A debug action has access to all of the data structures of the program that are in existence at the time it is invoked, as well as to the current status of the program's execution. Thus, the designer of a debug action has considerable flexibility in examining these various data structures and reporting their status to the user. In addition, the debug action may include a "break" statement, causing a temporary interruption of the program and allowing the programmer to enter interactive debugging commands.

The debugging commands include the standard operator's capabilities plus the ability to display and modify various aspects of the current computation either in terms of the source code or, if a more machine-oriented view is required, in terms of hexadecimal addresses and bytes. This language allows the user to view and change the values in data structures including the counting semaphores used to

implement Path Expressions, and to directly call subroutines and processes. Furthermore, the debugger allows the user to single step a statement at a time through any process currently active in the program. Finally, the user is able to formulate and modify the debug expressions and debug actions at any breakpoint.

Another important feature of the debugger is that all commands issued at breakpoints are stored in a journal. This allows the user to review these commands at a later time if necessary. In addition to this journal, the programmer may request a snap-shot or post-mortem dump for later analysis. This dump may be a symbolic listing of various data structures within the program or a crude hexadecimal image of memory.

Server Debugging

Entry/exit of servers usually represent major points in the execution of a program; hence, debug expressions may refer to these events. In addition, it will be important to be able to make inquiries of these various servers while at a breakpoint. On the other hand, when debugging a program, a server's internal operation is considered to be atomic and, therefore, invisible to the debugger. This allows the INTERRUPT PROCESSes typically found in device servers to continue to operate normally.

Indeed, it is these INTERRUPT PROCESSes that make debugging of servers special. In general, it is necessary to be able to take control of the entire node, not just a single program running on that node. The key difference is that the kernel itself must be suspended in order to allow the user to view the details of the processing of an interrupt or a sequence of interrupts. While this level of debugging should not predominate, it is vital that at least minimal support be available.

System debugging

Once the software system reaches the stage of operating across a network of nodes, it is vital that there be a debug node. This debug node will have access to all messages within the system [37], yet it will not appreciably change the timing of the messages. It simply records all of the messages (or all of the messages that have been designated as interesting) for later review or replay, or examines these messages looking for a user-specified pattern.

Debugger Implementation

While the generation of error-detection code is a significant contribution of the compiler to the debugging effort, an even more important contribution is the generation of tables that facilitate other aspects of the debugging system. The symbol table includes all CONST and TYPE information as well as the locations of all global, parameter, and local variables. The statement table provides the starting address of each statement in the software, any special information such as register usage, and a pointer into the source code. The path expression table describes all path expressions.

V. CONCLUDING REMARKS

Future aerospace vehicle computer systems which use distributed networks of computers to implement "cooperative" autonomous subsystems have requirements which distinguish them from their ground-based counterparts. Features of their operating systems are unique. Simply, two of the most important general requirements are time-critical performance and reliable networking. The network and operating system functional design specifications are tailored to meet these requirements.

Time-critical performance impacts a number of areas including the following:

1. Performance is enabled by using concurrency, that is, the use of parallel processing to gain speed. In order to take advantage of concurrency, the programmer must be provided with considerable freedom to design and manipulate local and remote services. This translates to programmer-controlled synchronization and recovery in contrast to structure imposed by either the programming language or the operating system. Concurrent language and adequate synchronization structure are needed to implement concurrency throughout the O/S support, service, and program layers of the system in a Higher Order Language without undue penalty.

2. Performance is enabled by proper file system structure and activity. In order to achieve the ability to read and store rapidly, an underlying real-time file system is implemented. The RT system is designed to incorporate normal file utilities and superimpose a time-shared structure above it. While such a file system may degrade TS performance during program development, the fact that it enhances time-critical performance is overriding.

3. Performance is enabled by compile-link-load time binding as opposed to runtime binding. While flexibility in many network TS systems is important, it is sacrificed for runtime efficiency here. The design demonstrates that not only can all normal binding take place before runtime, but much of the message traffic routing, porting, and virtual circuit assignments can be accomplished before execution begins. In fact, the only activity besides call-return pairs handled by the runtime system during normal execution is the dynamic thread of control needed for recovery.

4. Performance is enabled by time controlled message delivery. The proposed system with parallel bus and multipath connections provides a flexible, unique structure to handle message traffic. With the redundancy of message routes, critical message traffic can be handled by a bus specifically designated to provide time controlled delivery. With message virtual circuitry established prior to runtime, messages are handled with little system delay.

5. Performance is enabled by programmer freedom. This is a freedom fraught with frustration during program development and test. Distributed system reliability is a research area which to date has been attacked mainly by formal structure. However, the formal structure impacts performance. Hence, in a free environment, the ability to monitor, trace, control, resequence, restart, etc., activity across the network in order to detect and correct faults and/or to determine

correct execution is vital. This report provides design details for implementing this debug structure into the programming environment.

6. Performance is enabled by an integrated operating system as opposed to a layered structure. By integrating the common and unique node services at one level, the O/S support level; by differentiating between local and remote access; and by providing the node unique services with direct access to the node hardware and software structure, speed is enhanced and the need for buffering reduced. The price paid for this efficiency is that changes in functionality of the O/S components may not be transparent to other O/S support, service and application programs.

Reliable networking impacts a number of areas. The network operating system deals with the following:

1. Point 5 above, stresses that the absence of errors, including distributed sequencing and timing errors, is critical. In a system with the absence of control structure as presently designed, superior debugging tools are vital to reliable execution.

The network operating system does not deal with the object and action structure which are used for attaining reliable distributed sequencing and timing. It does, however, provide sufficient flexibility in language and O/S implementation that, if or when such structure becomes necessary, it can be incorporated. It also allows such structure to be readily implemented by system and application programmers and allows them to take advantage of any control structure built into specific service objects.

2. Reliable networking is enabled by redundant communication paths. The network system provides a parallel bus, multipath structure with superior redundancy characteristics. With this structure a wide range of message traffic loading can occur without serious degradation in controlled delivery time. If a failure of the time-critical message path should occur, the regular message path serves as an immediate backup. The regular message traffic system is then forced to find a new route. Hence, as long as a regular traffic route exists, the critical message route has a backup.

Finally, the network structure and operating system design are feasible. They provide a common structure and interface for building autonomous subsystems into a cooperative system and for incorporating extensibility, by easily permitting the addition of new nodes with new services.

VI. REFERENCES

1. Swingle, W. L.; McKay, C. W.: "Space Station Information Systems," Proc. of AIAA Symp. on Space Station, July 1983.

2. Garman, J. R.: "Forecasting Trends in NASA Flight Software Development Tools," AIAA Conference on Aerospace Computers, Oct. 1983.

3. Wittie, L. D.: "Communication Structures for Large Networks of Microcomputers," IEEE Trans. on Computers, Vol. C-30, No. 4, April 1981, pp. 264- 273.

4. Feldman, J. A.: "High Level Programing for Distributed Computing." Comm. of ACM, Vol. 22, No. 6, June 1979, pp. 353-363.

5. Wulf, W. A.; Levin, R.; Harbison, S. P.: HYDRA/C.MMP: An Experimental Computer System, McGraw Hill, N. Y., 1981.

6. Liskov, B.; Herlihy, M.: "Issues in Process and Communication Structure in Distributed Programs," 3rd Symp. on Reliability in Distributed Software & Data Base Systems, IEEE Comp. Soc. Press, Oct. 17-19, 1983, pp. 123- 132.

7. Cook, R.: "*MOD - A Language for Distributed Programing," IEEE Trans. on Software Engineering, Vol. SE-6, No. 6, Nov. 1980, pp. 563-571.

8. Campbell, R.: "Distributed Path Pascal," Distributed Computing Systems, Academic Press, London, 1983, pp. 191-223.

9. Wittie, L. D.; Van Tilborg, A.: "MICROS - A Distributed Operating System for MICRONET - A Reconfigurable Network Computer," Tutorial, Microcomputer Networks, ed H. A. Freeman & K. J. Thurber, IEEE Press, 1981, pp. 138-147.

10. Brownbridge, D. R.; Marshall, L. F.; Randell, B.: "The Newcastle Connection or UNIXes of the World Unite," Software Practice and Experience, Vol. 12, No. 12, Dec. 1982, pp. 1147-1162.

11. Bernstein, P.; Goodman, N.: "Concurrency Control in Distributed Data Base Systems," Computing Surveys, Vol. 13, No. 2, June 1981, pp. 185-221.

12. Allchin, J. E.: "A Suite of Robust Algorithms for Maintaining Replicated Data Using Weak Consistency Conditions," 3rd Symp. on Reliability in Distributed Software and Database Systems, IEEE Comp. Soc. Press, Oct. 17-19, 1983, pp. 47-55.

13. Internet Transport Protocols, Xerox System Integration Standard, Xerox Corp.; Stamford, Conn., 1981.

14. Courier: The Remote Procedure Call Protocol, Xerox System Integration Standard, Xerox Corp.; Stamford, Conn., 1981.

15. Moss, J. E. B.: "Nested Transactions: An Approach to Reliable Distributed Computing," Tech Rpt. MIT/LCS/TR-260, MIT, Cambridge, Mass., 1981.

16. Andrews, G. R.; Schneider, F. B.: "Concepts & Notions for Concurrent Programing," Computing Surveys, Vol. 15, No. 1, March 1983, pp. 3-43.

17. Lui, M. T.;Lian, R. C.: "Cells: An Approach to Design of a Fault- Tolerant Network Operating System," 3rd Symp. on Reliability in Distributed Software and Data Base Systems, IEEE Comp. Soc. Press, Oct. 17-19, 1983, pp. 163-172.

18. Liskov, B.; Scheifler, R.: "Guardians and Actions: Liguistic Support for Robust, Distributed Programs," Proc. of 9th Annual ACM Symp. on Princples of Programing Languages, Jan. 1982, pp. 7-19.

19. Campbell, R. H.; Randell, B.: "Error Recovery in Asynchronous Systems," UIUCDCS-R-83-1148, Univ. of Ill. at Urbana-Champaign, Urbana, Ill., Dec. 1983.

20. Jalote, P.; Campbell, R. H.: "Fault Tolerance Using Communicating Sequential Processes," UIUCDCS-R-83-1149, Univ. of Ill. at Urbana-Champaign, Urbana, Ill., Dec. 1983.

21. Kim, K. H.: "Approaches to Mechanization of the Conversation Scheme Based on Monitors," IEEE Trans. on Software Engineering, Vol. SE-8, No. 3, May 1982, pp. 189-197.

22. Wittie, L. D.; Curtis, R.: "Debugging Distributed Real-Time System," Workshop on Real-Time Operating Systems, Niagara Falls, N. Y., Aug. 1983.

23. Oppen, D. C.; Dalal, Y. K.: "The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment," ACM Trans. on Office Information Systems, Vol. 1, No. 3, July 1983, pp. 230-253.

24. Wittie, L. D.; Palumbo, M. J.; Frank, A. J.: "Overview of the Stand Alone Modula-2 System," Dept. of Computer Sci.; SUNY/Stony Brook, Stony Brook, N. Y., Draft Rept., June 6, 1983.

25. McKendry, M. S.: "Language Mechanisms for Context Switching and Protection in Level Structured Operating System," Ph.D. Thesis, Univ. of Ill. at Urbana-Champaign, Urbana, Ill., 1982.

26. Kolstad, R. B.: "Distributed Path Pascal: A Language for Programing Coupled Systems," Ph.D. Thesis, Univ. of Ill. at Urbana-Champaign, Urbana, Ill., 1982.

27. Wirth, N.: Modula-2, 2nd Ed. Tech. Rpt. Institut for Informatik ETH, No. 36, Zurich, 1982.

28. Wirth, N.: Programming in Modula-2, Springer Verlag, N. Y. 1982.

29. Ada Programing Language, ANSI/Mil-Std-1815A, 22 Jan. 1983.

30. Campbell, R. H.; Kolstad, R. B.: "An Overview of Path Pascal's Design and Path Pascal User Manual," SIGPLAN Vol. 15, No. 9, pp. 13-24.

31. Wei, A. Y.: "Real-Time Programming With Fault Tolerance," PH. D. Thesis, Univ. of Ill. at Urbana-Champaign, Urbana, Ill., 1981.

32. Feyock, S.: "Definition of the Operational Semantics of Ada Tasking," Final Rept. Grant NAG-1-62, College of William & Mary, Williamsburg, Va., 1981.

33. Wittie, L. D.; Curtis, R.: "Naming in Distributed Systems," SUNY/Stony Brook, Stony Brook, N. Y.; C. S. Tech. Rept. 83/056, Sept. 1983.

34. Ellis, C. S.; Floyd, R. A.: "The ROE File System," 3rd Symp. on Reliability in Distributed Software & Data Base Systems, IEEE Comp. Soc. Press, Oct. 17-19, 1983, pp. 175-181.

35. Hoare, C. A. R.: " Communicating Sequential Processes," Comm. of ACM, Vol. 21, No. 198, August 1978, pp. 666-677.

36. Bates, P.; Wileden, J.: "EDL: A Basis for Distributed System Debugging Tools," Proc. 15th Hawaii Int. Conf. on System Sciences, pp. 1986-93.

37. Curtis, R.; Wittie, L. D.: "BUGNET: A Debugging System for Parallel Programming Environments," Proc. IEEE 3rd Int. Conf. Dist. Comp. Sys., Oct. 1982, pp. 394-399.

38. LeBlanc, R.: "Interactive Debugging of Distributed Programs," Symp. on High-Level Debugging, March 1983.

39. Rietschote, H. F.: "Debugging in a CHILL Oriented Program Development System," Phillips' Telecom. Industries, Neth., 2nd CHILL Conference, Chicago, Ill., March 1983.

40. Richie, D. M.; Thompson, K.: "The UNIX Time-Sharing System," Comm. of ACM, Vol. 17, No. 197, July 1974, pp. 365-375.

41. CCITT, "CHILL Language Definition," CCITT Recommendation Z.200, Nov. 1980.

42. Allchin, J. E.; McKendry, M. S.: "Support for Objects and Actions in Clouds: Status Report," Tech. Rept. GIT-ICS-83/11, Georgia Inst. of Tech., Atlanta, Ga., May 1983.

43. Postel, J. B.: "Internetwork Protocol Approaches," IEEE Transactions on Communications, Vol. COM-28, No. 4, April 1980.

44. Bruegee, B.; Hibbard, P.: "Generalized Path Expressions: A High Level Debugging Mechanism," Preliminary Draft, ACM 0-89791-11-3/83/008/0034, 1983 pp. 34-44.

45. Berman, W. J.: "Development of a Prototype Multi-processor Interactive Software Invocation System," NASA CR – 172210, Cont. No. NASI-16985, Sept. 1983.

46. Tanenbaum, A. S.: Computer Networks, Prentice-Hall, Englewood Cliffs, N. J.: 1981, Ch 1.

47. Spector, A. Z.: "Performing Remote Operations Efficiently on a Local Computer Network," Comm. of ACM, Vol. 45, No. 4, April 1982, pp. 246-260.

48. Birrell, A. O.; Nelson, B. J.: "Implementing Remote Procedure Calls," ACM Trans. on Comp. Sys., Vol. 2, No. 1, Feb. 1984, pp. 39-59.

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA TM-85784		

4. Title and Subtitle	5. Report Date
AN OPERATING SYSTEM FOR FUTURE AEROSPACE VEHICLE COMPUTER SYSTEMS	April 1984
	6. Performing Organization Code 505-37-03-01
7. Author(s) Edwin C. Foudriat, W. J. Berman*, Ralph W. Will and W. L. Bynum**	8. Performing Organization Report No.
9. Performing Organization Name and Address NASA Langley Research Center Hampton, VA 23665	10. Work Unit No.
	11. Contract or Grant No.
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, DC 20546	13. Type of Report and Period Covered Technical Memorandum
	14. Sponsoring Agency Code

15. Supplementary Notes

*Advanced Programming Techniques, Inc., Charlottesville, Virginia
**College of William and Mary, Williamsburg, Virginia

16. Abstract

The requirements for future aerospace vehicle computer operating systems are examined in this paper. The computer architecture is assumed to be distributed with a local area network connecting the nodes. Each node is assumed to provide a specific functionality. The network provides for communication so that the overall tasks of the vehicle are accomplished.

The O/S structure is based upon the concept of objects. The mechanisms for integrating node unique objects with node common objects in order to implement both the autonomy and the cooperation between nodes is developed.

The requirements for time-critical performance and reliability and recovery are discussed. Time-critical performance impacts all parts of the distributed operating system; e.g., its structure, the functional design of its objects, the language structure, etc. Throughout the paper the tradeoffs--concurrency, language structure, object recovery, binding, file structure, communication protocol, programmer freedom, etc.--are considered to arrive at a feasible, maximum performance design.

Reliability of the network system is considered. A parallel multipath bus structure is proposed for the control of delivery time for time-critical messages. The architecture also supports immediate recovery for the time-critical message system after a communication failure. Techniques to enable programmer control of recovery are incorporated into the system design. Because finding and fixing errors in concurrent, distributed systems are extremely difficult, debug features are provided at the operating system, user interface level to enable the programmer to monitor and control tasks both within and between nodes.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Networks, operating system, distributed computer systems, real-time computer, aerospace computer	Unclassified - Unlimited Subject Category 62

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	36	A03

For sale by the National Technical Information Service, Springfield, Virginia 22161